Computational Structures in Data Science

SQL

Week 7, Summer 2024. 7/30 (Tues)

Lecture 23

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- "Official" course evaluation survey released
  - https://course-evaluations.berkeley.edu/Berkeley/
  - Filling this out will be a huge help for the Data/DSUS department with regard to course development, so please fill this out :)
- Project02 ("Ants") due this Thursday (8/1)!

Announcements: Final Exam

- Final exam "primary" time: **Wednesday August 7th, 3PM – 5 PM PST**
- Final exam scheduling form released
  - Gradescope: "(Urgent) Final Exam Scheduling"
  - Due: **Wednesday July 31st 11:59 PM PST**
  - Please fill this out ASAP! For more info, see Ed post.
  - DSP students: you should have received an email for a separate Final exam scheduling form. Please fill this out – if you didn't receive an e-mail, let us know!
- Final exam will have the same style as the Midterm
  - Zoom+Gradescope

Computational Structures in Data Science

Filtering in SQL

Berkeley
UNIVERSITY OF CALIFORNIA

Filtering rows - where

- Set of Table records (rows) that satisfy a condition

```
select [columns] from [table] where [condition] order by [order] ;
```

```
In [5]: cones.select(['Flavor', 'Price'])
```

Out[5]:

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	5.25
strawberry	5.25
bubblegum	4.75
chocolate	5.25

```
sqlite> select * from cones where Flavor = "chocolate";
ID|Flavor|Color|Price
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
6|chocolate|dark brown|5.25
```

```
cones.where(cones["Price"] > 5)
```

:

ID	Flavor	Color	Price
3	chocolate	dark brown	5.25
4	strawberry	pink	5.25
6	chocolate	dark brown	5.25

SQL:

```
sqlite> select * from cones where Price > 5;
ID|Flavor|Color|Price
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
6|chocolate|dark brown|5.25
```

SQL Operators for predicate

- use the WHERE clause in the SQL statements such as SELECT, UPDATE and DELETE to filter rows that do not meet a specified condition

SQLite understands the following binary operators, in order from highest to lowest precedence:

```
||
*    /    %
+    -
<<   >>   &    |
<    <=   >    >=
=    ==   !=   <>   IS   IS NOT   IN   LIKE   GLOB   MATCH   REGEXP
AND
OR
```

Supported unary prefix operators are these:

```
-    +    ~    NOT
```

Approximate Matching: LIKE [Docs]

- LIKE compares text to a *pattern*
  - *Case-Insensitive* by default. Means 'a' and 'A' are the same.
- Allows "wildcards" that match any character.
- % means "zero or more" characters at this "spot" in the pattern
- Examples:

```
'abc' LIKE 'abc' → true
'abc' LIKE 'a%' → true
'abc' LIKE '%b%' → true -shortcut for "does abc contain b?"
'b' LIKE '%b%'  → true
'abc' LIKE 'c' → false
```


Summary

- SQL a declarative programming language on relational tables
  - largely familiar to you from data8
  - create, select, where, order, group by, join
- Databases are accessed through Applications
  - e.g., all modern web apps have Database backend
  - Queries are issued through API
    - Be careful about app corrupting the database
- Data analytics tend to draw database into memory and operate on it as a data structure
  - e.g., Tables

Summary – Part 1

```
SELECT  <col spec> FROM <table spec> WHERE <cond spec>
    GROUP BY <group spec> ORDER BY <order spec> ;
```

```
INSERT INTO table(column1, column2,...)
       VALUES (value1, value2,...);
```

```
CREATE TABLE name ( <columns> ) ;
```

```
CREATE TABLE name AS  <select statement> ;
```

```
DROP TABLE name ;
```

Computational Structures in Data Science

SQL: Aggregations

Berkeley
UNIVERSITY OF CALIFORNIA

Aggregations are Powerful & Common!

```
SELECT date, COUNT(*)
FROM users
WHERE date > "2024-06-01"
GROUP BY date;
```

This query means: From the `users` table, fetch all rows whose `date` column is > `current_date`, group them by `date`, and count how large each group is.

Note: query results are also a table!

user_name	date
"bob"	2024-06-01
"alice"	2024-06-01
"louis"	2024-06-02
"alyssa"	2024-06-03
...	...

`users` table

date	count
2024-06-02	136
2024-06-03	257
2024-06-04	326
2024-06-05	167
...	...

There are 257 rows with date=2024-06-03

Query results

Grouping and Aggregations

- The `GROUP BY` clause is used to group rows returned by SELECT statement into a set of summary rows or groups based on values of columns or expressions.
- Apply an aggregate function, such as SUM, AVG, MIN, MAX or COUNT, to each group to output the summary information.

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	5.25
strawberry	5.25
bubblegum	4.75
chocolate	5.25

```
cones.group('Flavor')
```

Flavor	count
bubblegum	1
chocolate	3
strawberry	2

```
sqlite> select count(Price), Flavor from cones group by Flavor;
count(Price)|Flavor
1|bubblegum
2|chocolate
2|strawberry
```

```
cones.select(['Flavor', 'Price']).group('Flavor', np.mean)
```

Flavor	Price mean
bubblegum	4.75
chocolate	5.08333
strawberry	4.4

```
sqlite> select avg(Price), Flavor from cones group by Flavor;
avg(Price)|Flavor
4.75|bubblegum
5.0|chocolate
4.4|strawberry
```

Grouping and Aggregations

aggregator function (average)

Group keys (can be more than one!)

```
sqlite> select avg(Price), Flavor from cones group by Flavor;
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

GROUP by Flavor

```
# Flavor, Price
strawberry, 3.55
strawberry, 5.25

# Flavor, Price
chocolate, 4.75
chocolate, 5.25
chocolate, 5.25

# Flavor, Price
bubblegum, 4.75
```

avg(Price)

```
# Flavor, Price
strawberry, (3.55
+ 5.25) / 2

# Flavor, Price
chocolate, (4.75 +
5.25 + 5.25) / 3

# Flavor, Price
bubblegum,
(4.75) / 1
```

```
# Flavor, avg(Price)
strawberry, 4.4
chocolate, 5.0833
bubblegum, 4.75
```

avg(Price) is the result of computing the average Price within each group

Group by: "Bare" columns

- Caution: when using "GROUP BY", all selected columns MUST be one of:
  - (1) included in "GROUP BY"
  - Or
  - (2) in an aggregation function

```
sqlite> select flavor, avg(price), color from cones
group by flavor;

# flavor, avg(price), color
bubblegum|4.75|pink
chocolate|5.08333333333333|light brown
strawberry|4.4|pink
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

SQL **arbitrarily** chose one of the "chocolate" group Color values: "light brown" or "dark brown". Weird!

It's confusing to have a query where it's not clear what the output will be. Hence the recommendation to put selected columns in either "GROUP BY" clause OR in an aggregation function.

Note: in many other SQL engines, it's an error to have a "bare" column.

For more info on sqlite3's "bare columns", read this interesting (and colorful) thread: https://sqlite.org/forum/forumpost/4c8a673560d7999a

Unique & DISTINCT values

```
select DISTINCT [columns] from [table] where [condition] order by [order];
```

```
sqlite> select distinct Flavor, Color from cones;
strawberry|pink
chocolate|light brown
chocolate|dark brown
bubblegum|pink
sqlite>
```

```
In [8]: cones.groups(['Flavor', 'Color']).drop('count')
```

Out[8]:

Flavor	Color
bubblegum	pink
chocolate	dark brown
chocolate	light brown
strawberry	pink

```
In [7]: np.unique(cones['Flavor'])
Out[7]: array(['bubblegum', 'chocolate', 'strawberry'], dtype='<U10')
```

Computational Structures in Data Science

SQL: Joins

Berkeley
UNIVERSITY OF CALIFORNIA

Joining tables

- Two tables are joined by a comma to yield all combinations of a row from each (aka "Cartesian Product")
  - select * from sales, cones;

```
create table sales as
    select "Baskin" as Cashier, 1 as TID union
    select "Baskin", 3 union
    select "Baskin", 4 union
    select "Robin", 2 union
    select "Robin", 5 union
    select "Robin", 6;
```

Cashier	TID
Baskin	1
Robin	2
Baskin	3
Baskin	4
Robin	5
Robin	6

```
sales.join('TID', cones, 'ID')
```

TID	Cashier	Flavor	Color	Price
1	Baskin	strawberry	pink	3.55
2	Robin	chocolate	light brown	4.75
3	Baskin	chocolate	dark brown	5.25
4	Baskin	strawberry	pink	5.25
5	Robin	bubblegum	pink	4.75
6	Robin	chocolate	dark brown	5.25

```
sqlite> select * from sales, cones;
Baskin|1|1|strawberry|pink|3.55
Baskin|1|2|chocolate|light brown|4.75
Baskin|1|3|chocolate|dark brown|5.25
Baskin|1|4|strawberry|pink|5.25
Baskin|1|5|bubblegum|pink|4.75
Baskin|1|6|chocolate|dark brown|5.25
Baskin|3|1|strawberry|pink|3.55
Baskin|3|2|chocolate|light brown|4.75
Baskin|3|3|chocolate|dark brown|5.25
Baskin|3|4|strawberry|pink|5.25
Baskin|3|5|bubblegum|pink|4.75
Baskin|3|6|chocolate|dark brown|5.25
Baskin|4|1|strawberry|pink|3.55
Baskin|4|2|chocolate|light brown|4.75
Baskin|4|3|chocolate|dark brown|5.25
Baskin|4|4|strawberry|pink|5.25
Baskin|4|5|bubblegum|pink|4.75
Baskin|4|6|chocolate|dark brown|5.25
Robin|2|1|strawberry|pink|3.55
Robin|2|2|chocolate|light brown|4.75
Robin|2|3|chocolate|dark brown|5.25
Robin|2|4|strawberry|pink|5.25
Robin|2|5|bubblegum|pink|4.75
Robin|2|6|chocolate|dark brown|5.25
Robin|5|1|strawberry|pink|3.55
Robin|5|2|chocolate|light brown|4.75
Robin|5|3|chocolate|dark brown|5.25
Robin|5|4|strawberry|pink|5.25
Robin|5|5|bubblegum|pink|4.75
Robin|5|6|chocolate|dark brown|5.25
Robin|6|1|strawberry|pink|3.55
Robin|6|2|chocolate|light brown|4.75
Robin|6|3|chocolate|dark brown|5.25
Robin|6|4|strawberry|pink|5.25
Robin|6|5|bubblegum|pink|4.75
Robin|6|6|chocolate|dark brown|5.25
```

Joins

- Joins combine two tables
- A "cross product" or full join gives *all combinations*
- **This is often not useful!**
- So, we can do an *inner join* where we "combine" rows only on some logical identifier, like an "id"
  - Often this is called a "foreign key" or a reference to an object in another table.

Inner Join

Suppose that the `sales, cones` tables are set up such that sales.cone_id corresponds to the cones.id.
Ex: sales row `Baskin,1,2` means that Cashier=Baskin sold a cone with cone_id=2, aka `1,chocolate,light brown,4.75`
Then, to determine the Flavor, Color and Price of all sales made, we do this join:

```
SELECT * FROM cones, sales WHERE cone_id = cones.id;
```

When column names conflict we write: `table_name.column_name` in a query.

```
Id|Flavor|Color|Price|Cashier|id|cone_id
1|strawberry|pink|3.55|Baskin|3|1
1|strawberry|pink|3.55|Robin|6|1
2|chocolate|light brown|4.75|Baskin|1|2
2|chocolate|light brown|4.75|Baskin|4|2
2|chocolate|light brown|4.75|Robin|5|2
3|chocolate|dark brown|5.25|Robin|2|3
```

```
# Cashier, id, cone_id
Baskin|1|2
Baskin|3|1
Baskin|4|2
Robin|2|3
Robin|5|2
Robin|6|1
```

`sales` table

```
# id, Flavor, Color, Price
1|strawberry|pink|3.55
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
5|bubblegum|pink|4.75
6|chocolate|dark brown|5.25
7|vanilla|white|5.0
```

`cones` table

Tip: sales.cone_id is called a "foreign key" that refers to cones.id!

Putting It All Together:

- Which of our cashiers sold the highest value of ice cream?
- First we need to find which cones were sold by whom, then we SUM() the results!


```
sqlite> SELECT Cashier, SUM(Price) as 'Total Sold' FROM sales,
cones WHERE sales.cone_id = cones.id GROUP BY Cashier;
Cashier|Total Sold
Baskin|13.3
Robin|13.8
```

Question: Write a SQL query that can tell us which Cashier sold the highest value of ice cream.

Queries within queries

- Any place that a table is named within a select statement, a table could be computed
  - As a sub-query

```
select TID from sales where Cashier is "Baskin";

select * from cones
    where ID in (select TID from sales where Cashier is "Baskin");

sqlite> select * from cones
   ...>     where ID in (select TID from sales where Cashier is "Baskin");
ID|Flavor|Color|Price
1|strawberry|pink|3.55
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
```


Computational Structures in Data Science

SQL: CREATE and INSERT and UPDATE

(THIS IS NOT TESTED IN 88C!)

Berkeley
UNIVERSITY OF CALIFORNIA

CREATE TABLE

- SQL often used interactively
  - Result of select displayed to the user, but not stored
- Can create a table in many ways
  - Often may just supply a list of columns without data.
- Create table statement gives the result a name
  - Like a variable, but for a permanent object

```
CREATE TABLE [name] AS [select statement];
```

SQL: creating a named table

```
CREATE TABLE cones AS
    select 1 as ID, "strawberry" as Flavor, "pink" as Color,
3.55 as Price union
    select 2, "chocolate", "light brown", 4.75 union
    select 3, "chocolate", "dark brown", 5.25 union
    select 4, "strawberry", "pink",5.25 union
    select 5, "bubblegum", "pink",4.75 union
    select 6, "chocolate", "dark brown", 5.25;
```

Notice how column names are introduced and implicit later on.

Inserting new records (rows)

- A database table is typically a shared, durable repository shared by multiple applications

```
INSERT INTO table(column1, column2,...)
       VALUES (value1, value2,...);
```

```
sqlite> insert into cones(ID, Flavor, Color, Price) values (7, "Vanila", "White", 3.95);
sqlite> select * from cones;
ID|Flavor|Color|Price
1|strawberry|pink|3.55
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
5|bubblegum|pink|4.75
6|chocolate|dark brown|5.25
7|Vanila|White|3.95
sqlite>
```

```
cones.append((7, "Vanila", "White", 3.95))
cones
```

ID	Flavor	Color	Price
1	strawberry	pink	3.55
2	chocolate	light brown	4.75
3	chocolate	dark brown	5.25
4	strawberry	pink	5.25
5	bubblegum	pink	4.75
6	chocolate	dark brown	5.25
7	Vanila	White	3.95

UPDATING new records (rows)

- If you don't specify a WHERE, you'll update all rows!

```
UPDATE table SET column1 = value1, column2 =
value2 [WHERE condition];
```

Summary (and https://code.cs61a.org SQL demo!)

```
SELECT  <col spec> FROM <table spec> WHERE <cond spec>
    GROUP BY <group spec> ORDER BY <order spec> ;
```

```
INSERT INTO table(column1, column2,...)
       VALUES (value1, value2,...);
```

```
CREATE TABLE name ( <columns> ) ;
```

```
CREATE TABLE name AS  <select statement> ;
```

